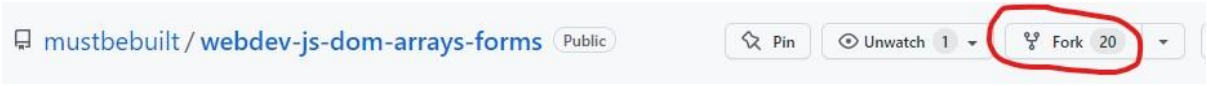
JAVASCRIPT - THE DOM, ARRAYS, FORMS

FORK THE GITHUB REPO

The files for this lab are already in a GitHub repo found at:

https://github.com/mustbebuilt/webdev-js-dom-arrays-forms

Fork this repo into a one of your own.



CONFIGURE GIT

In Visual Studio Code ensure you have entered your GitHub credentials. You may have already done this from a previous lab. You can check with:

```
git config user.name
git config user.email
```

If you haven't entered your credentials, this is done in terminal with:

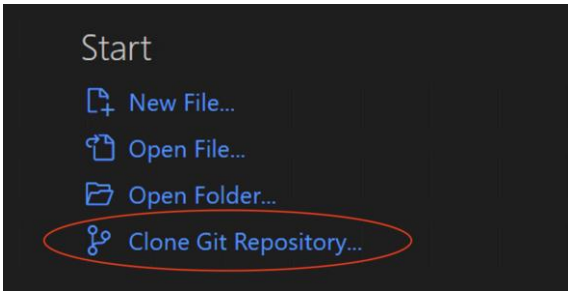
```
git config --global user.name "Joe Bloggs"
```

… and then add your email:

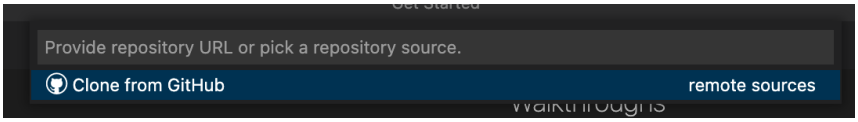
```
git config --global user.email b123456@my.shu.ac.uk
```

CLONE THE REPOSITORY

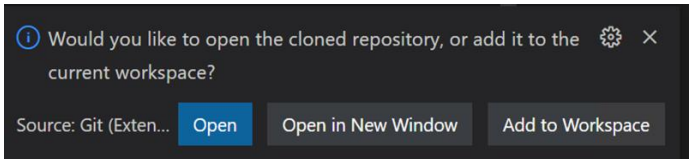
Use the option on the Splash Screen to Clone Git Repository.



Select **Clone from GitHub** and your repos should appear as you type your username into the prompt.



Visual Studio Code will then prompt you to select a target folder on your machine as the location to save your local copy of the repository. Choose somewhere sensible you will be able to easily locate again. Once cloned you are prompted to **Open** the cloned repository. Open it as prompted.



You should now see the project files. The repo contains several HTML files, *index.html*, *aboutUs.html* and *exchangeRate.html*, along with images and two stylesheet files.

FILES FROM THE ZIP

Alternatively, you can download the ZIP file for this week's lab and extract the files and folders.

To add the labs files for this week, drag the folder required into the Visual Studio Code workspace.

JAVASCRIPT AND THE DOM

Javascript is a client side scripting language. In web development it is mainly used to manipulate the DOM (Document Object Model).

The DOM is the HTML tree that makes up the HTML page. Javascript can be used to identify elements in the DOM and then be used to amend them. This is done by either adding HTML or CSS to change the appearance of the page. Javascript can also be used to insert new DOM elements, ie add in new HTML elements not originally in the file, and if needed, it can also remove DOM elements.

ADDING JAVASCRIPT TO A WEB PAGE

When adding Javascript to a web page there is a similar range of options to those you are hopefully familiar with from CSS. In CSS, styles can be added inline (directly to HTML element), at a document level (using the `<style>` tag) or loaded from an external stylesheet most frequently with the `<link>` tag.

With Javascript it can be added inline such as:

```
<body onload="alert('Just a Test')">
```

However, this technique is generally not recommended (as with inline CSS) as it makes code difficult to manage.

Javascript can also be added on a document basis. This is done with the `<script>` tag. Inside of this can be added Javascript.

```
<script>
alert('Fired from the script');
</script>
```

Finally, Javascript can be held in an external JS file and referenced by the document that wishes to use it. This is also done using the `<script>` tag as follows:

```
<script src="js/myScript.js"></script>
```

The value of the `src` attribute is a path to a JS file in the file structure. This can be a relative or absolute path as necessary. Inside the *js/myScript.js* will just be Javascript.

The `<script>` tag can be added anywhere in the document. However, it is good practise to either:

1. Place the `<script>` at the top of the document as a child of the `<head>` of the HTML file.
2. Place the `<script>` near to the bottom of the document before the closing `</body>` tag.

Either approach is valid but if you do place the `<script>` at the top of the document you should ensure you code only runs when the document has loaded (more later).

ALERTS AND CONSOLES

Open the file *index.html*. Here is a standard HTML document with the usual mix of images and text.

Add the following before the closing `</head>` tag:

```
<script>
alert('Hello World');
</script>
```

Save and refresh you page in the browser. You should see a Javascript alert box.

Note: *Javascript alert boxes are just simple dialogue boxes. Although tempting to use them to provide user feedback they are little untidy and often associated by users as 'errors'. It is much better for users to add messages to the DOM rather than as alerts.*

Values in the DOM can be extracted using a range of methods and properties. Most of these relate to a Javascipt object known as `document`. This represents an object that represents the DOM.

Change the alert above to the following:

```
<script>
console.dir(document);
</script>
```

In Google Chrome use the developers tools to locate the 'console'. This is an essential part of the toolkit when working with Javascript.

The `console.dir()` command in the above snippet tells the console to output the content of the document object.

When adding Javascript to your pages you will find the console very useful. As well as `console.dir()` which is used to debug objects you can also use other commands such as `console.info()` and `console.log()`.

Other browser do have similar tools but this author rates the Chrome tools as currently the most useful.

GETTING TO KNOW THE DOM

We saw above that there is a `document` object. This object has a large range of methods and properties. These can be used to target elements in the DOM, and then extract or change their value.

The easiest of the many methods used to identify DOM elements looks for an element by id and is called `document.getElementById()`.

Once an element is targeted we could retrieve its contents using the property `innerHTML`.

Notice that the `<h1>` has an id attribute:

```
<h1 id="myHeading">My Demo Site</h1>
```

To retrieve the value inside of this `<h2>` we can use:

```
document.getElementById("myHeading").innerHTML
```

We could try to console this value ie:

```
console.info(document.getElementById("myHeading").innerHTML);
```

However, in the console we will get an error:

This is because we have tried to reference a DOM element before the browser has had chance to load the DOM. There are various solutions here:

1. Add an event that will trigger the code when the DOM has loaded.
2. Move the `<script>` to the bottom of the document before the closing `</body>` as the DOM will have loaded at that point.
3. Use the `defer` attribute of the `<script>` tag. This ensures the Javascript is only executed when the page has finished parsing. This modern approach may cause issues with some older browsers.

All are perfectly valid. Let's see how we could accomplish this with the first option.

Leave the `<script>` in the `<head>` of the document but place all the code inside of a function as follows:

```
<script>
function init(){
      console.info(document.getElementById("myHeading").innerHTML);
}
</script>
```

We now have a function called `init()`. Next we'll create an event to call the `init()` function when the document has loaded. Earlier, it was suggested that inline Javascript is not recommended. However, this is one use-case that is commonly used by developers.

Amend the `<body>` tag as follows:

```
<body onLoad="init()">
```

Save and test your file. You should now see the console outputting the contents of the `<h1>` tag.

Another solution for this issue is to place the code at the bottom of the file. With this second approach there is no need to have the `onload` event attached to the `<body>` or to wrap your code inside of a function.

Remove the `onLoad` event from the `<body>` tag and move the `<script>` tag to before the closing `</body>` tag.

You should now also remove the `init()` function so the code appears as:

```
console.info(document.getElementById("myHeading").innerHTML);
```

This could also be amended to use an immediately invoked function expression as follows:

```
(function(){
      console.info(document.getElementById("myHeading").innerHTML);
})();
```

It is common to store Javascript in an external for re-use. Create a *js* folder in your application structure and create a file called *main.js*.

Move the Javascript from inside the `<script>` into this external file. Replace the `<script>` tag in *index.html* with a reference to the external file.

```
<script src="js/main.js"></script>
```

Finally, to use the modern refer technique you can move the `<script>` back into the `<head>` but at defer as follows:

```
<script src="js/main.js" defer></script>
```

CHANGING THE DOM

To change the value inside of the `<h1>` identified above, we can use `innerHTML` as this is a property that can both be read and set. Change your by removing the console call and replacing it with:

```
document.getElementById('myHeading').innerHTML = "Your Name";
```

Save and test your file and you should see that 'Your Name' appear in the DOM.

As well as changing the content of a HTML element we can change its attributes. This done with a method of the document object called `setAttribute()`. The `setAttribute()` method takes two parameters, firstly the attribute to add and secondly its value.

We would like to add the class 'currentPage' to the first `<li>` in the menu. This class is already in the CSS for this file and will simply underline the targeted element. As such the `setAttribute()` called would be:

```
setAttribute('class', 'currentPage');
```

As this element does not have an ID we need an alternative approach to select it. To do this we could use the method `document.querySelector()`. This takes as parameters a CSS selector and will return the first match found. As the `<li>` is the first one in the list we could use a CSS selector of:

```
nav ul li
```

As such add the following to the Javascript code:

```
document.querySelector('nav ul li').setAttribute('class', 'currentPage');
```

Save and test your file. The 'Home' link should now appear underlined as a result of the class.

To show the power of this we'll change the link on the home button such that it points to *http://www.google.co.uk*.

To do this add:

```
document.querySelector('nav ul li a').setAttribute('href',
'http://www.google.co.uk');
```

Test the page. You may want to remove this after proving it is possible!

***Note**: This isn't really something you'd be likely to do but does illustrate why developers need to be wary of a hacker attack known as Cross Site Scripting (XSS) where a hacker will attempt to run their Javascript on your page.*

EVENTS

Events are a crucial part of Javascript development, so much so that Javascript is often described as an 'event driven' scripting language. There are a huge amount of events that Javascript can react to including click, onload, mousemove, mouseover, submit and even gestures like touchstart.

Events can be added a number of different ways.

Inline Events

Inline evens are where the event is placed directly in the HTML code eg:

```
<p onClick="console.info('hi')">Test Event</p>
```

We have already seen a use of this with the `<body onload="init()">`. However, apart from this common use case inline events are discouraged as they make code hard to manage.

Note: With inline events the event itself is case insensitive, so all of these are equally valid: onclick, onClick, ONclick.

DOM Events

DOM events reference DOM elements and set an event property to point at a function. This function contains the code triggered as a result of the event and is known as an event handler. For example, in the code below the property `onclick` is used to reference an anonymous function that acts as the event handler.

```
document.getElementById('myTestEvent').onclick =  function(){
      console.info('hi');
}
```

With DOM event the event property is case sensible and is in lowercase (not camelCase), so `onClick` should be invalid.

The example above uses an anonymous (ie unnamed function). Equally functions can be referenced by name eg:

```
document.getElementById('myTestEvent').onclick = myFunction;
function myFunction(){
      console.info('hi');
}
```

Notice that the `onclick` property points at the named function, it does not call it. As such there is no need for the parenthesis `()` after the function name.

USING THE ADDEVENTLISTENER() METHOD

The recommended way to work with events is the `addEventListener()` method. It is more flexible than the previous two approaches and is part of the DOM Events specification.

At its most basic the `addEventListener()` method takes two parameters. These are an event type and then an event handler.

The event can be any number of different events such as 'click', 'mouseover' or 'submit'

The event handler is a function, either named or anonymous, that is called when the event is triggered. An example of an anonymous function as the event handler would be:

```
document.getElementById('myTestEvent').addEventListener('click',
function(){
	console.info('hi');
})
```

It could also reference a named function eg:

```
document.getElementById('myTestEvent').addEventListener('click',
myFunction);
function myFunction(){
	console.info('hi');
}
```

As with the DOM event the `addEventListener` registers the function to call, thus the lack of parenthesis.

THE EVENT OBJECT

One useful feature of both DOM events and `addEventListener` is the ability to pass the event object itself to the event handler function. This is done by default but to expose this object we need to add a parameter to the event handler function. You can call this anything you like (within Javascript naming rules). Here we are using `ev` as it is nice short way to refer to an event.

```
document.getElementById('myTestEvent').addEventListener('click',
function(ev){
	console.dir(ev);
})
```

Open the console and you will see that you have a `MouseEvent`. There are a lot of interesting properties and methods such `target` which indicates the target of the event.

CHALLENGE 1: CHANGING THE BACKGROUND COLOUR OF THE PAGE

In the *index.html* there are a number of `<span>` tags designed to allow users to change the background colour of the file.

Create an event for each of the red, blue and green options to add the appropriate CSS class to the body tag. To do so you'll need to use `querySelector()` and `setAttribute()`.

Create an event for the reset option to set the background colour back to white. To do so use the `removeAttribute()` method.

The easiest way to achieve the above is to would be to use an event such as:

```
document.querySelector(".red").addEventListener('click', function(ev){
        document.querySelector('body').setAttribute('class', "redBack");
   })
```

You could then repeat the above for each of the colours as well as the reset to default.

ARRAYS IN JAVASCRIPT

Arrays are created in Javascript with either:

```
let myArray = new Array();
```

Or more efficiently with just:

```
let myArray = [];
```

In either case values can be added as a comma separated list ie:

```
let myArray = new Array('cat', 'dog', 'mouse');
```

Or

```
let myArray =  ['cat', 'dog', 'mouse'];
```

As arrays represent more complex variable add them to the console with `console.dir()`.

```
console.dir(myArray);
```

Array values are accessed via their index number eg:

```
myArray[0];// cat
myArray[2];// mouse
```

Indexing starts from zero.

Loops are commonly used to extract values from an array:

```
for(let i=0; i<3; i++){
      console.info(myArray[i]);
}
```

The array property `length` can be used to find the number of values in an array.

CREATING AN ARRAY OF IMAGES

In the file *index.html* on line 26 is an `<img>` with an ID of `myImages`. We'll create an array to change the image displayed.

In the *js/main.js* file create an array as follows:

```
let imageAr = ['images/view1.jpg', 'images/view2.jpg', 'images/view3.jpg',
'images/view4.jpg', 'images/view5.jpg', 'images/view6.jpg'];
```

Any of these images can be assigned to the image `src` attribute with for example:

```
document.getElementById('myImages').setAttribute('src', imageAr[2]);
```

To have the images rotate we could use the Javascript `setInternal()` method. This will call a given method/function at a set interval defined in milliseconds.

To test this, create a function as follows:

```
function chgImage(){
      console.info('Called');
}
```

and then reference the function with a `setInterval()`:

```
setInterval(chgImage, 2000);
```

This will change the value every 2 seconds.

CHALLENGE 2: FIX AND EXTEND THE IMAGE ROTATOR

The image rotator has a bug in that it will run out of images. We need a counter to help keep track of which image is displayed and switch back to the first one if we reach the end.

Declare a counter variable and increment it in the `chgImage()` function.

Use conditional logic and the `length` property to loop the selected image back to the beginning of the array is the last image is been displayed.

Change the delay on the `setInterval` to 4000 (4 seconds).

Try adding a manual way for the user to progress the images by clicking on the current image.

DOCUMENT.QUERYSELECTOR() VS DOCUMENT.QUERYSELECTORALL()

The `document.querySelectorAll()` method works like the `document.querySelector()` method in that it uses CSS to select on element in the DOM. The difference is that:

- `document.querySelector()` returns the first found match;
- `document.querySelectorAll()` returns all the matches.

The `document.querySelectorAll()` return the matches in the form of a `NodeList`. This behaves like an array of nodes.

The following will create a `NodeList` of the HTML nodes used to change the background colour:

```
let colourButtons = document.querySelectorAll(".colPicker")
console.dir(colourButtons)
```

Notice in the console they appear as a `NodeList`. It looks and behaves like an array, so can be looped.

CHALLENGE 3: REFACTOR THE BACKGROUND COLOUR CHANGING CODE

Now you have done some work with arrays, revisit the background colour changing code.

The previous approach would work but it is repetitive. Could you refactor it?

Tip: you could use `document.querySelectorAll()` to return a `NodeList` of all the `.colPicker` nodes. A `NodeList` is an array like structure, so you could loop it.

As you loop the list you could add an event listener to each of the `.colPicker` options.

```
for(let i=0; i < colourButtons.length; i++){
    colourButtons[i].addEventListener("click", chgColour)
}
```

In the above code snippet the NodeList is looped and each node in the list has the event Listener attached to it. When any .colPicker element is clicked the event listener will capture the event and in this example call the function `chgColour()`.

You will need to create that function ie:

```
function chgColour(ev){
    console.dir(ev.target.classList)
}
```

Notice how the `event` is passed to the event hander as `ev`.

Investigate the `classList` property. You could use this to retrieve the colour picked in an event handler method and match it to the appropriate css classes.

```
let colourPicked = ev.target.classList[0] + "Back"
```

With the above snippets refactor the colour picking code.

WORKING WITH THE WINDOW OBJECT

The `window` object in Javascript represents the browser window. As such it can be used to access information about the current window. As with the `document` object we can review it via the console with:

```
console.dir(window)
```

Adding the above to our *main.js* file would produce an output as follows:

Note the `location` object. This provides information about the current path. Amend the console statement to:

```
console.dir(window.location)
```

This will reveal the properties and methods of the `location` object and reveal a property of `href`. This is useful for determining the filename of the current page.

The location object can therefore be used to retrieve information about the current page such as its URL (href) with `window.location.href`.

To only retrieve the filename from the string that `window.location.href` generates we can use two other built in methods, one for strings and one for arrays.

Javascript strings have a method of `split()`. This will divide the string into an array based on a delimiting/separator character.

By calling `split()` against the string returned by `window.location.href` we can create an array of the folders and files from the address by using the `/` (forward slash) as the separator.

Try adding:

```
let url = window.location.href;
console.dir(url.split('/'));
```

We now have an array of values. Notice the last value is the filename currently been viewed. We can then use the array method `pop()` to extract and return the last value in the array.

In Javascript method calls like this can be 'chained' ie called one after another to work on the return value of the former.

Therefore, the filename can be retrieved with:

```
let url = window.location.href;
let filename = url.split('/').pop();
```

CHALLENGE 4: CUSTOMIZE THE NAVIGATION TO THE PAGE

We previously added the `.currentPage` class to the `li` of the homepage. It would make sense if this could be done uniquely for every page.

The list of HTML pages that appear in the navigation could be retrieved with:

```
let allLinks = document.querySelectorAll('nav ul li');
```

You could then loop and retrieve the `href` attribute value of each `a` link and compare it to the current page retrieved from the `window.location.href` value.

Where there is a match the class of `.currentPage` could be applied to the relevant list item.

WORKING WITH FORMS

HTML forms are often associated with Javascript, as it can be used to retrieve form values and to perform tasks such as form validation.

Inside the file *exchangeRate.html* there is a basic HTML form. We'll use this form to build a simple currency convertor. Create a second external Javascript file in *js/exchange.js*.

Add `<script>` tag to the bottom of the file just before the closing `</body>` to load this external file as follows:

```
<script src="js/exchange.js"></script>
```

Inside *js/exchange.js* add the following immediately invoked function expression:

```
(function(){
      // logic here
})();
```

For this application we'll need five variables. There are three values needed for the calculation. That is the value in pounds to be converted, the exchange rate and the value in euros. Declare these inside of the immediately invoked function:

```
let pricePounds;
let exchangeRate;
let priceEuros;
```

Then we could also use a variable representing the HTML form and a HTML element used to feedback to the user.

Notice that the `<form>` has an ID. As such we can reference it with `document.getElementById`. We also have a `<p>` of ID `msg` that we'll use to feedback to the user. This is currently set to `display:none` is the CSS. Add two new variables as follows:

```
let convertForm = document.getElementById('myForm');
let msg = document.getElementById('msg');
```

HANDLING FORM SUBMISSION

When the submit button is pressed the HTML form would normally be submitted to the action of the form. In this application we don't want that to happen as it would by default cause a page refresh. We would like a form submission to perform the currency conversion.

With Javascript we can listen out for events that occur and capture them. In this example we listen out for the form submission and capture it to do our currency conversion. We'll also stop the form from submitting which is it default behaviour.

To capture the submit event we'lll use `addEventListener()`. This will also allow use to expose the event object.

Add the following:

```
convertForm.addEventListener('submit', function(ev){
      ev.preventDefault();
})
```

The above adds an event listener to the form and listens for when it is submitted. When it is submitted the event triggers the anonymous function.

As we have passed the event object to the function and exposed it as `ev` we can call a method of the event object `preventDefault()` that will stop the form submitting in the normal way. That way we can perform our calculation logic and display the result to the user without a page refresh.

RETRIEVING THE FORM VALUES

To retrieve the values from the form we can use the property `value`.

A value in pounds is entered by the user into this field:

```
<input type="text" id="pounds">
```

We can therefore retrieve that value with:

```
document.getElementById('pounds').value;
```

The `value` property, like `innerHTML`, can be used to get and set the value.

With HTML forms all values are returned as strings not numbers, so even if the user entered 12 it is a string value of '12'.

To convert the value to a number we can use the method `parseFloat()`. This will take a value and convert it into a floating point (decimal). If it cannot convert the value then it will return `NaN` (not a number).

***Note**: As well as parseFloat() Javascript has other methods such as parseInt() and parseString() to help with data type conversion.*

Inside of the event handler we could now add:

```
priceEuros = 0;
exchangeRate = 0.87;
pricePounds = parseFloat(document.getElementById('pounds').value);
```

If the value entered by the user is not a number, we need to provide a suitable message. The `isNaN()` method can be used to see if a value is a number.

```
if(isNaN(pricePounds)){
      msg.style.display = "block";
      msg.innerHTML = "You must enter a number";
      msg.setAttribute('class', 'error');
}
```

If the variable `pricePounds` is not a number (`isNaN`) then the HTML of ID `msg` is displayed. This is done using `style` object of the DOM element and setting its `display` property to `block`.

This adds an inline CSS style. When initially styling a file, inline styles are usually frowned upon but as this style is added via Javascipt as part of an inaction with the user is it a perfectly valid approach. We also add a suitable error message via `innerHTML` and use `setAttribute()` to add the class of `error` which is already defined in the CSS.

If the number is valid add an else block to the above if condition. Inside this else block we'll check to see if the number entered is zero.

```
if(pricePounds === 0){
      msg.style.display = "block";
      msg.innerHTML = "You must enter more than zero."
      msg.setAttribute('class', 'error');
}else{
      priceEuros = pricePounds * exchangeRate;
      msg.style.display = "block";
      priceEuros = priceEuros.toFixed(2);
      msg.innerHTML = "You will get &euro;"+priceEuros;
      msg.removeAttribute('class');
}
```

If the value is zero then a suitable error message is displayed. If not then the calculation is performed and the result displayed back to the user. Note the user of the HTML entity to ensure consist display of the euro sign.

RESETING THE INPUT FIELD WITH USING THE EVENT OBJECT

Finally, we'll reset the input field for the pounds value when the user places their focus in it.

As a new event add:

```
document.getElementById('pounds').addEventListener('focus', function(){
      this.value = "";
      msg.innerHTML = "";
      msg.removeAttribute('class');
})
```

This uses a `focus` event. A focus event is triggered when a form field receives the focus of the cursor. Once triggered we clear the error message and its error message status.

The `this.value` uses the `this` keyword which refers to the current context. Here this refers to the DOM element that has the event handler attached to it ie:

```
document.getElementById('pounds')
```

As such here, `this.value` refers to the value of input field.

USING THE EVENT OBJECT

Alternatively, the above could be achieved by exposing the event object in the event handler function as follows:

```
document.getElementById('pounds').addEventListener('focus', function(ev){
      ev.target.value = "";
      msg.innerHTML = "";
      msg.removeAttribute('class');
})
```

This uses a `focus` event. A focus event is triggered when a form field receives the focus of the cursor.

The event is exposed as `ev` and then the value extracted via `ev.target.value`. The target property of the event object refers to the target of the event ie:

```
document.getElementById('pounds')
```

Once triggered we clear the error message and its error message status.

CHALLENGE 5: IMPROVE THE EXCHANGE RATE CALCULATOR

Can you amend the logic of the Exchange Rate Calculator to work both as £ to € and € to pounds?

BONUS IMAGE SLIDER

A common feature of web sites is an image slider. This is an enhanced version of the image rotator from earlier in the lab.

These codepens may help you with the concepts used. This first Pen shows the logic so you can toggle different parts of the CSS and Javascript to see how the banner is constructed:

https://codepen.io/mustbebuilt/pen/VwqaoWm

As the logic uses the Modulus % operator to loop the index value this CodePen should help you understand this refactoring trick:

https://codepen.io/mustbebuilt/pen/zYyBOvK

In the *index.html* replace the:

```
    <p>
    <img src="" alt="Winter Gardens" id="myImages">
    </p>
```

With:

```
<div class="slider">
 <div class="slides">
  <img src="images/banner-image-1.jpg" alt="Silver Ball Fountains">
  <img src="images/banner-image-2.jpg" alt="Winter Gardens Roof External">
  <img src="images/banner-image-3.jpg" alt="Winter Gardens Roof Internal">
  <img src="images/banner-image-4.jpg" alt="Crucible Theatre">
  <img src="images/banner-image-5.jpg" alt="Now Then Graffiti">
  <img src="images/banner-image-6.jpg" alt="Winter Gardens Palm Trees">
</div>
<button id="prevBtn">Previous</button>
<button id="nextBtn">Next</button>
</div>
```

Add the CSS to the *mobile.css* as follows:

```
.slider {
  position: relative;
  max-width: 100% ;
  min-width: 300px;
  margin: auto;
  overflow: hidden;
}

.slides {
  display: flex;
  transition: transform 0.4s ease-in-out;
}

.slides img {
```

```
    max-width: 100%;
      height: auto;
}
```

This places the images inside a container with a max-width but also an `overflow:hidden` property.

Without this the images would appear as follows:

The `overflow:hidden` property hides the images outside of the `div.slider`. The aim of the Javascript will be to offset the `div.slides` inside of the `div.slider` so that a different image is presented. This will be done with the CSS `transform: translateX(someValue)` property.

Notice the CSS for the `div.slides` use a CSS transition which as the `transform` value is changed via Javascript will transition the image accordingly.

We will create a new JS file of *basicBanner.js* and attach it to your index.html with the `<script>` tag. Add the following Javascript code to create the image banner.

```
   (function(){
const prevBtn = document.getElementById('prevBtn');
   const nextBtn = document.getElementById('nextBtn');
   const slides = document.querySelector('.slides');
})();
```

Here, we're using `document.getElementById` and `document.querySelector` to select the HTML elements with the IDs `prevBtn` and `nextBtn`, and the div of class `slides`.

Initialize a variable called `currentIndex` which keeps track of the index of the currently displayed slide. It starts at 0.

```
let currentIndex = 0;
```

Add click event listeners to the "Previous" and "Next" buttons. When these buttons are clicked, the corresponding functions (`prevSlide` and `nextSlide`) are executed (we will create these later).

```
prevBtn.addEventListener('click', prevSlide);
nextBtn.addEventListener('click', nextSlide);
```

Add a function to change the position of the slides by setting the `transform` property of the `.slides` element. By multiplying the index by 100%, we're shifting the slide container horizontally to display the desired slide.

```
function showSlide(index) {
     slides.style.transform = `translateX(-${index * 100}%)`;
   }
```

When the "Next" button is clicked, this function is called. It updates the `currentIndex` by incrementing it and handling the wrapping when reaching the last slide. It also calls `showSlide()` to display the updated slide.

```
function nextSlide() {
     currentIndex = (currentIndex + 1) % slides.children.length;
     showSlide(currentIndex);
   }
```

Note the use of the `%` modulus operator. This returns the remainder of dividing the first value by the second. For example 5 % 4 returns 1. This is a useful in this context as when the `currentIndex` reaches the same value of the number of images we want it to return 0. As such 5 % 5 returns 0, in effect resetting the `currentIndex`.

When the "Previous" button is clicked, this function is called. It calculates the new `currentIndex` by decrementing it and handling the wrapping around when reaching the beginning of the slides. Then it updates the displayed slide using `showSlide()`.

```
function prevSlide() {
     currentIndex = (currentIndex - 1 + slides.children.length) %
slides.children.length;
     showSlide(currentIndex);
   }
```

Here again the `%` modulus operator is used. If decrementing `currentIndex` to a value of 0 we find the logic 4 % 5 returns 4.

The `autoSlide` function calls `nextSlide` to move to the next slide. The `setInterval` function is used to repeatedly call `autoSlide` every 5000 milliseconds (5 seconds), creating an automatic slide change effect.

```
function autoSlide() {
     nextSlide();
   }
   setInterval(autoSlide, 5000);
```

By combining these elements, you've created the basic functionality of an image slider. The `currentIndex` keeps track of the displayed slide, and the `showSlide` function updates the slider's position to show the desired slide. The "Previous" and "Next" buttons trigger functions to move between slides, and the optional `setInterval` adds an automatic slide change feature.

You could take this example further by:

- Adding navigation 'dots';
- Replacing the header background image with the slider
- Loading the images source and alternative text from an array.